A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range

Frithjof Blomquist, Werner Hofschuster and Walter Krämer

Bergische Universität Wuppertal, Scientific Computing/Software Engineering, Gaußstraße 20, D-42097 Wuppertal, Germany
{Blomquist, Hofschuster, Kraemer}@math.uni-wuppertal.de
http://www.math.uni-wuppertal.de/~xsc

Abstract. A so called staggered precision arithmetic is a special kind of a multiple precision arithmetic based on the underlying floating point data format (typically IEEE double format) and fast floating point operations as well as exact dot product computations. Due to floating point limitations it is not an arbitrary precision arithmetic. However, it typically allows computations using several hundred mantissa digits.

A set of new modified staggered arithmetics for real and complex data as well as for real interval and complex interval data with very wide exponent range is presented. Some applications show the increased accuracy of computed results compared to ordinary staggered interval computations. The very wide exponent range of the new arithmetic operations allows computations far beyond the IEEE data formats.

The new arithmetics would be extremly fast, if an exact dot product was available in hardware (the fused accumulate and add instruction is only one step in this direction).

Key words: staggered correction, multiple precision, C-XSC, interval computation, wide exponent range, reliable numerical computations, complex interval functions
AMS classification: 65G20, 65G30, 65Y99, 37M99, 30-04

1 Introduction

Staggered correction arithmetics [1, 20, 21, 32, 28, 33, 4, 6] are based on exact dot product computations of floating point vectors [10]. (Please refer to [28] for some historical remarks on the implicit/explicit usage of staggered numbers [31, 9, 1, 18, 26, 27, 21]). A staggered correction number is given as the (exact) sum of the components of a list of floating point numbers (components of a floating point vector). Thus the maximum precision of arithmetical operations depends strongly on the exponent range of the floating-point screen. Staggered correction numbers close to the underflow range are typically not as accurate as numbers with larger exponents. For the IEEE double precision format [3] the range for positive numbers is about 4.9E-324 to 1.7E+308. Thus, up to 630 (decimal) mantissa digits may be handled using staggered precision variables.

Dagstuhl Seminar Proceedings 08021
Numerical Validation in Current Hardware Architectures
http://drops.dagstuhl.de/opus/volltexte/2008/1445

To get high accuracy in numerical calculations, (intermediate) underflow and/or overflow situations must be avoided, i.e. an appropriate scaling of the operands of arithmetic operations is necessary. To this end we represent our modified staggered number as a pair (e, x). We refer to such pairs as extended staggered numbers. The integer e denotes an exponent with respect to the base 2 and x denotes an ordinary staggered number (i. e. a vector of floating point numbers). The value v of the modified staggered variable (e, x) is $v = (e, x) := 2^e \cdot \text{sum}(x)$. Here, $\text{sum}(x)$ means the exact sum of all floating point components of the staggered variable x. The pair-representation allows the handling of very small and very large numbers: 1.3E-487564, 4.1E9999999, or numbers with even larger exponents may be used in numerical calculations. We are aware of several test cases, where multiple-precision calculations using computer algebra packages like Maple and/or Mathematica fail, whereas our extended staggered software returns the expected results, see e.g. Subsection 5.2.

Our new package [6] offers real interval and complex interval arithmetic operations and also a rather complete set of mathematical functions for the new data types `lx_interval` (extended real staggered intervals) and `lx_cinterval` (extended complex staggered intervals). The trigonometric functions, the inverse trigonometric functions, the hyperbolic and the inverse hyperbolic functions as well as several other functions are provided for (rectangular) extended complex staggered intervals.

Some details of the arithmetic operations as well as some details on the implementation of elementary transcendental mathematical functions will be discussed in this paper. Applications (logistic equation, limit calculations, complex Interval Newton method) will be presented to demonstrate the ease of use and to show the superior behavior of the new arithmetics over the more traditional one. We will also discuss hardware requirements needed to get the staggered arithmetic working extremely fast.

The source code of the new package will be distributed under the GNU general public license. Up to now, it is an independent supplement to our C-XSC library. The source code will be made available online, see
`http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html`

2 (Extended) Staggered Data Types

Real, complex, real interval and complex interval staggered data types are realized as arrays of floating point numbers [20, 28, 6]. E.g. a real staggered variable is represented by an array of floating point values. The (exact) sum of these floating point numbers is the numerical value of the staggered number. The implementation of arithmetic operations for staggered numbers uses extensively the possibility to compute the value (the exact sum of its floating point components) of a staggered number error free. But exact summation is a special case of exact dot product computations. As we will see in a moment, this operation is also extensively used to implement staggered operations.

Some general remarks are appropriate: Accumulation of numbers is the most sensitive operation in floating point arithmetic [10]. By this operation scalar products of floating point vectors, matrix products etc. can be computed without any error in infinite precision arithmetic, making an error analysis for those operations superfluous. Many algorithms applying this operation systematically have been developed. For oth-

ers the limits of applicability are extended by using this additional operation. Furthermore, the exact dot product speeds up the convergence of iterative methods (cited from [23, 24]). XSC languages (e.g. C-XSC [17, 16]) provide exact dot products via software simulation (hardware support should increase the computation speed by 2 orders of magnitude, again, see [24]). Computing $x \cdot y$ for floating point vectors x, and y in C-XSC results in the best possible floating point result (exact mathematical result rounded to the nearest floating point number; correctly rounded result). Using the C-XSC data type `dotprecision` (so called long accumulator), the user can even store the result of dot products of floating point vectors with even quintillions of components without any error.

Let us introduce staggered numbers in a more formal way [20, 28, 6]. An ordinary staggered interval number x is given by

$$x := \sum_{i=1}^{n-1} x_i + [x_n, x_{n+1}] = \sum_{i=1}^{n-1} x_i + X.$$

Here, all x_i are floating-point numbers. The lower bound $\underline{x}$ of x is given by

$$\underline{x} := \sum_{i=1}^{n-1} x_i + x_n$$

and the upper bound $\overline{x}$ by

$$\overline{x} := \sum_{i=1}^{n-1} x_i + x_{n+1}.$$

Thus, for n equal 1 the staggered interval x collapses to the ordinary interval $X = [x_1, x_2]$. Because different staggered numbers usually have different numbers of floating point components, we indicate their individual lengths by n_x, n_y and so on. For the staggered number x from above, its length n_x is defined to be equal n. An extended real staggered interval (with length n) is given by

$$v = (e, x) = \left(e, \quad \sum_{i=1}^{n-1} x_i + [x_n, x_{n+1}] \right) := 2^e \cdot \sum_{i=1}^{n-1} x_i + [x_n, x_{n+1}].$$

To be more readable we avoid the explicit use of this representation whenever possible. Instead we just give some hints, how (automatic) scalings are done.

In the following subsections we describe the realization of arithmetic operations for (extended) staggered numbers x and y [28, 6]. The resulting (extended) staggered number is denoted by z. To simplify our presentation we assume that both operands x, y are aleady scaled properly. To simplify the presentation we omit the explicit use of the scaling factor introduced for the extended staggered data types [6]. Only in the case of division we add some remarks on an appropriate scaling of numerator and/or denominator.

The operands of an arithmetic operation may have different staggered length n_x, n_y, respectively. The staggered length n_z of the result z can be prescribed by the user of the package using the global variable `stagprec`. The resulting staggered (interval)

number z has to be computed in such a way that it contains all corresponding point results for arbitrary point arguments taken from the (staggered interval) operands x and y.

2.1 Addition/subtraction and multiplication

The (exact) result of these operations is always representable using two dotprecision variables (lower and upper bound are computable as exact scalar products of floating point vectors). Errors (overestimation) may only occur due to underflow situations when the content of the dotprecision variables is converted back to an array of floating point numbers (i.e. to a staggered interval number).

Let us consider the multiplication $x * y$ of two real staggered numbers $x = \sum x_i$ and $y = \sum y_i$, x_i, y_i being floating point numbers. We have to compute $x * y = \sum x_i * \sum y_i$. Thus, the exact result can be written in the equivalent form $\sum \sum x_i \cdot y_j$, i.e. as a dot product of length $n_x \cdot n_y$ of two vectors with floating point components $\in \{x_1, x_2, \ldots x_{n_x}, y_1, y_2, \ldots y_{n_y}\}$. This real number can be stored error free in a dotprecision variable. To get the final result, this intermediate (exact) result is to be stored as a sum of floating point values (i.e. as a staggered number). Please note that dotprecision values may or may not be representable as a sum of floating point numbers. This is due to the fact that a dotprecision value may have a much larger or much smaller exponent than ordinary floating point numbers allow. Thus, the resulting staggered representation z of the product $x * y$ is in general only an approximation to the exact intermediate value stored during the computation of the dot product. The accuracy of z is limited by the exponent range of the underlying floating point number system. Of course, when implementing staggered interval operations the errors introduced when going from an exact intermediate result to a staggered representation has to be taken into account. This can be done easily using directed rounded conversions whenever the content of a dotprecision variable has to be converted to an ordinary floating point number.

2.2 Division

Up to now, we are able to compute bounds for the results of our operations in two dotprecision variables and then round these to an interval staggered format. This can no longer be carried out conveniently in the case of the *division* of two staggered correction intervals $x = \sum_{i=1}^{n_x} x_i + X$ and $y = \sum_{i=1}^{n_y} y_i + Y$. Rather, we will apply an iterative algorithm computing successively the n_z real components z_i of the quotient x/y.

In order to compute this approximation $\sum_{i=1}^{n_z} z_i$, we start with $z_1 = \square m(x) \not\square \square m(y)$; here $m(a)$ represents a point selected in a, e.g. the midpoint and $\square$ is the rounding to the floating point screen S. Now, we proceed inductively: if we have an approximation $\sum_{i=1}^{k} z_i$, we can compute a next summand z_{k+1} by use of

$$z_{k+1} = \square \Big(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{k} y_i z_j \Big) \not\square \square(m(y)), \tag{1}$$

where the numerator is computed exactly using a dotprecision variable and is rounded only once to S. The division is performed in ordinary floating point arithmetic.

As in the previous operations, this iteration guarantees that the z_i do not overlap, since the defect (i.e. the numerator in (1)) of each approximation $\sum_{i=1}^{k} z_i$ is computed with only one rounding.

Now, the interval component Z of the result z may be computed as follows:

$$Z = \diamondsuit \Big(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{n_z} y_i z_j + X - \sum_{j=1}^{n_z} z_j Y \Big) \not\diamondsuit \diamondsuit(y), \tag{2}$$

where $\diamondsuit$ is the rounding to an enclosing interval in IS.

It is not difficult to see that $z = \sum_{i=1}^{n_z} z_i + Z$ as computed from (1) and (2) is a superset of the exact range $\{\xi/\eta \mid \xi \in x, \eta \in y\}$; in fact, for all $\alpha \in X, \beta \in Y$ we have the identity:

$$\frac{\sum\limits_{i=1}^{n_x} x_i + \alpha}{\sum\limits_{i=1}^{n_y} y_i + \beta} = \sum_{j=1}^{n_z} z_i + \frac{\sum\limits_{i=1}^{n_x} x_i + \alpha - \sum\limits_{i=1}^{n_y} \sum\limits_{j=1}^{n_z} y_i z_j - \sum\limits_{j=1}^{n_z} z_j \beta}{\sum\limits_{i=1}^{n_y} y_i + \beta}.$$

An interval evaluation of this expression for $\alpha \in X$ and $\beta \in Y$ shows immediately that the exact range of x/y is contained in $\sum_{j=1}^{n_z} z_j + Z$, which is computed using (1) and (2).

Now, it is clear how to get the result z for the division of two staggered interval variables x/y by the following three computation steps:

1. $z_1 := m(x) \not\boxslash m(y)$
2. compute real parts z_{k+1} from (1) for $k = 0, \ldots, n_z - 1$
3. compute interval part Z according to (2)

(3)

At this point we will discuss a proper scaling of the numerator x and/or denominator y (see [6] for more details). Let us assume for a moment that x and y are both very close to the overflow threshold (about 1e300 for double numbers). Then the result z of the division would be close to 1. Thus, the exponent of the leading term of the staggered number z would be zero. In this case only about the first 300 decimal digits of the exact result can be stored in an array of floating point numbers. Digits to the right of this leading part can not be accessed by floating point numbers due to underflow. Thus, even if numerator and denominator are point intervals we can not expect more than about 320 correct digits of the staggered result.

But we can do better. The general procedure is: Scale the numerator to be close to the overflow threshold maxreal and scale the denominator to be close to the square root of maxreal (about 1E150 for double numbers). Denote the scaled numbers by x and y, respectively. If so, the quotient of the scaled numbers will also be close to the square root of maxreal, i.e. its exponent will be close to 150. In this way we obviously can convert about 300 + 150, i.e. about 450 leading digits to floating point numbers. The correct scale factor of the result can be computed easily by integer addition/subtraction from the scale factors used to scale the numerator and denominator.

An algorithm for the *square root* can be obtained analogously as in the case of the division. We compute iteratively as follows the $z_i, i = 1, \ldots, n_z$ of the approximation part:

$$\begin{aligned} z_1 &= \sqrt{\square(x)} \\ z_{k+1} &= \square\Big(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{k} z_i z_j \Big) \;\boxslash\; (2z_1). \end{aligned} \tag{4}$$

This guarantees again that the z_i do not overlap since in the numerator of (4) the defect of the approximation $\sum_{i=1}^{k} z_i$ is computed with one rounding only. Now, the interval part Z is computed by use of

$$Z = \frac{\diamondsuit\Big(\sum\limits_{i=1}^{n_x} x_i - \sum\limits_{i,j=1}^{n_z} z_i z_j + X \Big)}{\sqrt{\diamondsuit(x)} + \diamondsuit\Big(\sum\limits_{i=1}^{n_z} z_i \Big)}. \tag{5}$$

As in the case of the division, it is easy to see that $\sum_{i=1}^{n_z} z_i + Z$ as computed from (4) and (5) is a superset of the exact range $\{\sqrt{\xi} \mid \xi \in x\}$; in fact, for all $\gamma \in X$ we have the identity:

$$\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} = \sum_{j=1}^{n_z} z_j + \frac{\sum\limits_{i=1}^{n_x} x_i + \gamma - \sum\limits_{i,j=1}^{n_z} z_i z_j}{\sqrt{\sum\limits_{i=1}^{n_x} x_i + \gamma} + \sum\limits_{j=1}^{n_z} z_j}.$$

Now, we see that the square root z of a staggered interval x can be computed by the following three steps very similar to the case of division:

1. $z_1 := \sqrt{\square x}$
2. compute real parts z_{k+1} from (4) for $k = 0, \ldots, n_z - 1$
3. compute interval part Z according to (5)

(6)

Before computing the square root as described above, the argument is scaled in the following way: Multiply the original argument by an even power of two, say by 2^{2n}, such that the scaled argument x comes close to maxreal. The correct scaling factor of the computed result then is 2^n. For a point interval argument we may expect up to about 450 correct digits.

3 Some Transcendental Elementary Functions

In this section we first give an overview to the implementation of the exponential function $\exp(x)$. The implementation of the natural logaritm $\ln(x)$ is described in more detail.

3.1 Exponential function

We only give an overview how the exponential function is realized [30]. The reader may refer to the source code to see the details.

The domain of the exponential using ordinary staggered intervals is only $|x| < 709.7$, whereas in case of the extended staggered interval data type is $|x| < 1488521882.0$.

To implement the exponential we use the relation $e^x \equiv \left(e^{x\cdot 2^{-n}}\right)^{2^n}$. More precisely we do the following:

- Choose n such that $|x| \cdot 2^{-n} \sim 10^{-9}$
- Perform a Taylor approximation: $e^{x\cdot 2^{-n}} \approx T_N(x \cdot 2^{-n})$
- Take into account absolute approximation error: $e^{x\cdot 2^{-n}} \subset U(x, N)$
- Perform a result adaptation: $e^x \subset U(x, N)^{2^n}$:

```
for (int k=1; k<=n; k++) U = sqr(U);
```

3.2 The Natural Logarithm

Let us now present the implementation of the natural logarithm $\ln(x)$ in more details.

There are two cases to be considered: X is close to the zero $x_0 = 1$ of the logarithm or x is sufficiently far away from this zero.

By calculating the logarithm function near the zero $x_0 = 1$, the problem arises that due to cancellation effects the rather small function values can only be included with a few correct decimal digits. To avoid this problem, we introduce the auxiliary function $ln1p(t) := ln(1 + t)$.

```
lx_interval Lnp1(const lx_interval& x)
```

is implemented in order to include the function values $\ln(1 + x)$, $x \approx 0$, of type `lx_interval` with sufficient accuracy. The algorithm is based on the well-known series expansion

$$\ln(1+x) = \zeta \cdot \sum_{k=0}^{\infty} \frac{2}{2k+1} \cdot (\zeta^2)^k, \quad \zeta := \frac{x}{2+x}, \quad x > -1, \quad |\zeta| < 1. \tag{7}$$

With the definitions $P(\zeta) := \sum_{k=0}^{\infty} \frac{2}{2k+1}\cdot(\zeta^2)^k,\ P_N(\zeta) := \sum_{k=0}^{N} \frac{2}{2k+1}\cdot(\zeta^2)^k,\ N \geq 0$ $\ln(1 + x)$ is approximated by

$$\ln(1+x) \approx \zeta \cdot P_N(\zeta),\ N \geq 0. \tag{8}$$

The absolute approximation error δ is defined by

$$\delta := |P(\zeta) - P_N(\zeta)| = \sum_{k=N+1}^{\infty} \frac{2}{2k+1} \cdot (\zeta^2)^k = \sum_{n=0}^{\infty} \frac{2}{2n+2N+3} \cdot (\zeta^2)^{n+N+1}$$

$$= (\zeta^2)^{N+1} \cdot \sum_{n=0}^{\infty} \frac{2}{2n+2N+3} \cdot (\zeta^2)^n \leq (\zeta^2)^{N+1} \cdot \frac{2}{2N+3} \cdot \frac{1}{1-\zeta^2}$$

and the last upper bound can further be estimated by

$$\delta := |P(\zeta) - P_N(\zeta)| \leq \frac{(\zeta^2)^{N+1}}{N+1}, \tag{9}$$

using the following inequalities:

$$\frac{2}{2N+3} \cdot \frac{1}{1-\zeta^2} < \frac{1}{N+1} \quad \Longleftrightarrow \quad \zeta^2 < \frac{1}{2N+3}.$$

The last one is guaranteed in all practical cases, because with e.g., $|x| \sim 10^{-7}$ also $|\zeta| \sim 10^{-7}$ is valid. As we shall see, the maximum polynomial degree is about $N_{max} = 42$, so that the upper bound $1/(2N+3)$ cannot become smaller than $1/(2N_{max}+3) = 1.14\ldots \cdot 10^{-2}$, i.e. $1/(2N+3)$ is surely greater than $\zeta^2 \sim 10^{-14}$.

We now consider an interval $\boldsymbol{x}$ with[1] $|\boldsymbol{x}| \ll 1$. Using $R := |\boldsymbol{x}/(2+\boldsymbol{x})| = |\boldsymbol{\zeta}|$, the absolute error δ is bounded by Δ

$$\delta = |P(\zeta) - P_N(\zeta)| \leq \frac{R^{2N+2}}{N+1} =: \Delta, \quad \forall \zeta \in \boldsymbol{\zeta} := \frac{\boldsymbol{x}}{2+\boldsymbol{x}}. \tag{10}$$

Concerning the approximation in (8) the next step is to determine an appropriate polynomial degree N. For $|\boldsymbol{\zeta}| \ll 1$ it holds $P(\zeta) \approx P_N(\zeta) \approx 2$ and with a given precision $p :=$ `stagprec` the absolute approximation error should be of order $2^{1-53\cdot p}$. Thus, together with (10) we require

$$\frac{R^{2N+2}}{N+1} = 2^{1-53\cdot p} \quad \Longleftrightarrow \quad R^{2N+2} = (N+1)\cdot 2^{1-53\cdot p}. \tag{11}$$

For a facile calculation of N, the right-hand side in (11) must be simplified futhermore. Together with `xi = li_part(x); ex = expo_gr(xi)` we have

$$|\boldsymbol{x}| \approx 2^{ex+\mathtt{expo}(\boldsymbol{x})} = 2^m, \quad \text{i.e.} \quad m := ex + \mathtt{expo}(\boldsymbol{x}).$$

Furthermore it holds: $R \approx |\boldsymbol{x}|/2 = 2^{m-1}$ with the consequence

$$2^{(m-1)\cdot(2N+2)} = (N+1)\cdot 2^{1-53\cdot p},$$

and due to $(N+1) = 2^{\log_2(N+1)}$ it follows

$$(m-1)\cdot(2N+2) = \log_2(N+1) + 1 - 53\cdot p$$
$$2N+2 = \frac{53\cdot p - 1 - \log_2(N+1)}{1-m}, \quad m \neq 1.$$

Caused by the requirement $|\boldsymbol{x}| \ll 1$, the condition $m \neq 1$ is surely complied. For the range $0 \leq N \leq 42$ it holds $0 \leq \log_2(N+1) < 5.43$, so that, with $p \geq 2$,

$$53\cdot p - 1 - \log_2(N+1) \approx 53\cdot p - 4$$

[1] $|\boldsymbol{x}| := \max\limits_{r\in\boldsymbol{x}}\{|r|\}$

is a good approximation for calculating N. So we get

$$2N+2 = \frac{53\cdot p-4}{1-m} \quad \Longleftrightarrow \quad N = \frac{53\cdot p-4}{2\cdot(1-m)} - 1.$$

To realize $N \geq 0$, the polynomial degree N is additionally increased by 1, and thus, for calculating N, we get the quite simple formula:

$$N = \frac{53\cdot p-4}{2\cdot(1-m)}, \quad m := ex + \texttt{expo}(\boldsymbol{x}), \quad N \geq 0. \tag{12}$$

The C-XSC function $\texttt{expo}(\boldsymbol{x})$ returns the exponent of the scaling factor of the extended staggered quantity of type `lx_interval`.

With a sufficient small $|\boldsymbol{x}|$ resp. with a sufficient great value $-m$ the evaluation of $2\cdot(1-m)$ in (12) will generate an overflow, which in case of $2\cdot(1-m) > 53\cdot p-4$ can be avoided by setting $N=0$.

Since the evaluation of $2\cdot(1-m)$ can possibly produce an overflow, the last condition $2\cdot(1-m) > 53\cdot p-4$ must further be transformed. It holds

$$\begin{aligned} 2\cdot(1-m) > 53\cdot p-4 \quad &\Longleftrightarrow \quad 2m < 6-53\cdot p \quad \text{resp.} \\ 2\cdot(ex+\texttt{expo}(\boldsymbol{x})) < 6-53\cdot p \quad &\Longleftrightarrow \quad \texttt{expo}(\boldsymbol{x}) < 3-\frac{53\cdot p}{2} - ex. \end{aligned}$$

The last inequality is valid, if

$$\texttt{expo}(\boldsymbol{x}) < 3-27\cdot p - ex. \tag{13}$$

Thus, if (13) is realized, N is set to zero, and otherwise N is calculated by (12). In either case an overflow is avoided.

The next step is to calculate the upper bound Δ of the absolute approximation error in (10), where Δ is defined as follows

$$\Delta = \begin{cases} R^2, & N=0, \\ \dfrac{R^{2N+2}}{N+1}, & N \geq 1. \end{cases} \tag{14}$$

The function `Lnp1(...)` ist implemented by:

```
lx_interval Lnp1(const lx_interval &x) throw()
// Calculating an inclusion of ln(1+x) for
// not too wide intervals,    |x| <= 1e-7;
{
    lx_interval res(0),z2,zeta,Ri,Two;
    l_interval xli;
    int N,ex,p;
    real m,expox;

    p = stagprec;
    xli = li_part(x);
    ex = expo_gr(xli);
```

```
    if (ex>-100000) // x <> 0
    {
        expox = expo(x);
        if (expox < 3-27*p-ex) N = 0;
        else
        {
            m = ex + expox;  // m = ex + expo(x);
            N = (int) _double( (53*p-4)/(2*(1-m)) );
        }
        // N: Polynomial degree, 0<=N<=42;
        zeta = x / (2+x);
        Two = lx_interval(2);
        Ri = lx_interval( Sup(abs(zeta)) );
        Ri = sqr(Ri);
        if (N==0)
            res = Two;
        else // N >= 1:
        {
            z2 = sqr(zeta); // z2 = zeta^2
            // Evaluating the polynomial:
            res = Two / (2*N+1);  // res = a_(N)
            for (int i=N-1; i>=0; --i)
                res = res*z2 + Two/(2*i+1);
            // Calculating the absolute approximation error:
            Ri = power(Ri,N+1)/(N+1);
        }
        // Implementing the approximation error:
        res += lx_interval(lx_real(0),Sup(Ri));
        res *= zeta;
    } // x <> 0;

    return res;
} // Lnp1(...)
```

Remarks:

1. Please note that due to the definition of the absolute approximation error δ in (9), the addition of its upper bound[2] Δ has to be done *before* the multiplication with ζ.
2. Due to $P_N(\zeta) < P(\zeta)$ the absolute approximation error is not included by `lx_interval(-Sup(Ri),+Sup(Ri))`, but, coupled with a more less interval inflation, by `lx_interval(lx_real(0),Sup(Ri))` instead.

With the help of the `Lnp1` function the logarithm function

```
lx_interval ln(const lx_interval &x);
```

[2] in the source code: $\Delta :=$ `Sup(Ri)`.

can now be implemented. In case of $\mathsf{x} \approx 1$, i.e. $|\mathsf{x} - 1| \leq 10^{-7}$, $\ln(\mathsf{x})$ is included by $\ln(\mathsf{x}) \subseteq \texttt{Lnp1}(\mathsf{x} \diamond 1)$ and in case of $|\mathsf{x} - 1| > 10^{-7}$ the following identity is used

$$\begin{aligned} \ln(x) &= \ln\left((x \cdot 2^{-n}) \cdot 2^n\right), \quad x \in \mathbb{R},\ n \in \mathbb{Z}, \\ &= \ln(t) + n \cdot \ln(2); \quad t := x \cdot 2^{-n} \in \mathbb{R}. \end{aligned}$$

Concerning the first reduced argument $t := x \cdot 2^{-n}, n \in \mathbb{Z}$ is calculated to realize $t \approx 1$ as well as possible, in order to evaluate $\ln(t)$ with an accuracy as high as possible by using function `Lnp1()`: $\ln(t) = \texttt{Lnp1}(t-1)$. Please keep in mind that the multiplication in $t = x \cdot 2^{-n}$ can be calculated error-free and quite effectively in the class `lx_interval`.

For a further improvement of $t \approx 1$ we use [30]

$$\ln(t) = 2^k \cdot \ln\left(\sqrt[2^k]{t}\right), \quad \text{with} \quad \lim_{k\to\infty} \sqrt[2^k]{t} = 1.$$

Thus, for a sufficient large $k \in \mathbb{N}$, the 2nd reduced argument $u := \sqrt[2^k]{t}$ lies sufficiently close to 1. However, for e.g. $k = 22$, the calculation of u, using the 2^k-th root function, is quite expensive. A much more efficient method is to evaluate the loop

```
u = t;
for (int j=1; j<=22; j++)
    u = sqrt(u);
```

where the last value `u` is a quite good approximation of $\sqrt[2^k]{t} \approx \mathsf{u}$. For the calculation of k, the exponent m of $x = M \cdot 2^m,\ 0.5 \leq M < 1$ has to be specified first.

Now choosing $n := m$, we get $t = M$, and, defining $t := 1 - \varepsilon$, it holds $0 < \varepsilon \leq 0.5$ and we get the quite good approximation

$$u = \sqrt[2^k]{t} = \sqrt[2^k]{1-\varepsilon} \approx 1 - \frac{\varepsilon}{2^k}.$$

For an efffective evaluation of $\ln(u) \approx \ln(1 - \varepsilon/2^k)$ we require

$$\begin{aligned} \frac{\varepsilon}{2^k} \leq 10^{-6} \quad &\Longleftrightarrow \quad \varepsilon \leq 2^k \cdot 10^{-6} \\ &\Longleftrightarrow \quad k \geq \frac{\ln(\varepsilon) + 6 \cdot \ln(10)}{\ln(2)}. \end{aligned}$$

In case of $\varepsilon \ll 1$, the above fraction becomes negative and $k = 0$ is set, i.e., the second argument reduction will fall away. In case of $k \geq 1$ it holds

$$\ln(x) = n \cdot \ln(2) + 2^k \cdot \ln(u), \quad t = x \cdot 2^{-n},\ u = \sqrt[2^k]{t},\ k = 1, 2, 3, \ldots \tag{15}$$

and $\ln(u) = \ln(1 + (u-1))$. In order to include $\ln(x)$, the right-hand side of (15) has to be evaluated by using intervals:

$$\ln(x) \in n \diamond <\ln(2)> \ \diamond 2^k \diamond \texttt{Lnp1}(\mathsf{u} \diamond 1),\ \ \forall x \in \mathsf{x}.$$

$< \ln(2) >$ denotes a guaranteed enclosure of $\ln(2)$ in high accuracy. It is predefined as a C-XSC staggered interval constant `Ln2_lx_interval()`. `u` is the result of the above `for` loop performed by an interval evaluation, and before this loop $t \in \mathbb{R}$ has to be substituted by $\mathtt{x} \diamond 2^{-n}$. The multiplication with 2^k can be realized effectively and error-free by use of the function `times2pown(...)`.

4 Complex Staggered (Interval) Operations and Complex Elementary Transcendental Functions

Extended complex staggered (interval) numbers are formed as pairs of two real extended staggered (interval) numbers in an obvious way (real and imaginary part, respectively). An extended complex staggered interval represents a rectangle in the complex plane with sides parallel to the axes. All basic operations as well as a large set of elementary transcendental functions are available for the new extended complex staggered interval data type [6]. More about complex interval functions may be found in [11, 12, 5, 7, 29].

4.1 Remark on Complex Division

By considering complex division we demonstrate the gain in accuracy when replacing the ordinary staggered arithmetic by the new extended staggered arithmeic. We restrict ourselves to the evaluation of the real part formula of a complex division. With

```
x= Rex + i Imx
y= Rey + i Imy
```

we have to compute

```
Re(x/y)= (Rex*Rey+Imx*Imy)/(Rey*Rey + Imy*Imy).
```

We perform this computation using ordinary staggered intervals and using extended staggered intervals.

In the following code snippet `l_interval` variables are ordinary staggered interval numbers:

```
stagprec= 30;

l_interval  Rex, Imx, Rey, Imy, q; //ordinary!
Rex= 1e150;  Imx= 1e150;  //x= 1e150  + 1e150 i
Rey= 1e-150; Imy=1e-150;  //y= 1e-150 + 1e-150 i

q= (Rex*Rey + Imx*Imy) / (Rey*Rey+Imy*Imy); //Re(x/y)

cout << "Re(x/y) ) = " << q << endl;
cout << RelDiam(q) << endl;

Output:
```

```
Re(x/y) ) = [9.99999999999999745402366E+299,
             9.99999999999999745402415E+299]
RelDiam(q): 4.94...e-24
```

Let us now perform the same computation using extended staggered intervals:

```
lx_interval  Rex, Imx, Rey, Imy, q; //extended!
Rex= 1e150;  Imx= 1e150;
Rey= 1e-150; Imy= 1e-150;

q= (Rex*Rey + Imx*Imy) / (Rey*Rey+Imy*Imy);

cout << RelDiam(q) << endl;

Output:
RelDiam(q): {-487, [ 1.1491393399, 1.1491393400]}
More than 480 correct decimal digits!
```

We see that in he first case only about 24 digits are correct, whereas in the second case (using the new data type `lx_interval`) we get more than 480 correct decimals. This very strong improvement is due to new operations performing optimal scalings automatically.

4.2 Complex Transcendental Functions

Availabe are the trigonometric functions, the inverse trigonometric functions, the hyperbolic functions and the inverse hyperbolic functions. For multivalued functions the principal branch is realized [11, 12, 7, 29]. Table 1 shows some further transcendental functions available for extended complex staggered intervals. For more details see [6].

5 Applications

In this section we demonstrate the ease of use and the power of staggered intervals delivering a special kind of a multiple precision arithmetic.

5.1 Dynamical System

Let us consider the simple dynamical system as given by the logistic equation:

$$x_{n+1} = a \cdot x_n \cdot (1 - x_n)\,, \;\; n \;\geq\; 0 \tag{16}$$

for some $a \in [0, 4]$ and $x_0 \in (0, 1)$.

Its Mean Value Form [28] (capital characters denote interval quantities) is:

$$\begin{gathered} X_{n+1} = a \cdot (y_n(1 - y_n) + (1 - 2X_n) \cdot (X_n - y_n)) \\ \text{with} \;\; y_n \;\approx\; \text{midpoint of } X_n. \end{gathered} \tag{17}$$

Table 1. Additional complex extended staggered interval functions

Complex Standard Functions of Type `lx_cinterval`		
Function Term	**C-XSC Name**	**Informations**
$\|z\| = \sqrt{(z.re)^2 + (z.im)^2}$	abs(z)	$\|z\| \subseteq$ abs(z)
z^2	sqr(z)	Inclusion of the squares
$\sqrt{z}$	sqrt(z)	Calculating the main branch
$\sqrt[n]{z}$	sqrt(z, n)	$2 \leq n \leq +2147483647$
arg(z)	Arg(z)	Analytic argument function
arg(z)	arg(z)	Not analytic function
ln(z)	Ln(z)	Analytic logarithm
ln(z)	ln(z)	Not analytic function
z^n	power_fast(z, n)	Fast power function
z^n	power(z, n)	Slower power function
z^p	pow(z, p)	lx_cinterval z; lx_interval p;
z^w	pow(z, w)	lx_cinterval z, w
e^z	exp(z)	lx_cinterval z

For parameter a close to 4 this system exhibits chaotic behavior. Ordinary floating point computations will deliver results which are completely wrong quantitatively, when compared with the true trajectory on which the computation began.

The naive interval approach resuts in the following source code:

```
stagprec = 39;
lx_interval a(3.75);
lx_interval x;

x= 0.5;
for (int k=1;;k++) {
  x= a*x*(1-x);
  if (Sup(x) > 1) break; //x must lie in [0, 1]
  if (k%50 == 0) cout << "x_" << k <<": " << x << endl;
}

Last output:
x_800: {-1, [ 4.3447661328, 4.3447661330]}
```

The source code for using the more sophisticated Mean Value form [28] is as follows:

```
//mean value form:
x= 0.5;
lx_interval y;
for (int k=1;;k++) {
  y= mid(x);
  x= a*( y*(1-y) + (1-2*x)*(x-y) );
  if (Sup(x) > 1) break;
  if (k%50 == 0) cout << "x_" << k <<": " << x << endl;
}

The last row of the generated output is
x_2750: {-1, [ 6.4898891410, 6.4898898170]}

Note that using ordinary intervals not even
100 time steps can be performed!
```

These computations show clearly that using the extended staggered operations, the number of time steps can be increased considerably (from about 100 to about 2750).

5.2 Limit calculation, powers with very high exponents

Let us try to compute some approximation to the Euler number e. More precisely, let us compute an enclosure of the approximation

$$(1+\frac{1}{n})^n \text{ with } n = 10^{600000000}$$

and let us see, how far

$$(1+\frac{1}{n})^n \text{ with } n = 400$$

is apart from the limit

$$e = \lim_{n\to\infty}(1+\frac{1}{n})^n\,.$$

Note that in the first calculation the exponent n, written as a decimal number, has six-hundred million and one decimals. The following source code does the job:

```
#include "../lx_imath.hpp"
using namespace std;
using namespace cxsc;

int main() {
   stagprec= 5;
   int sixHundredMillion(600000000);
```

```
   lx_interval n, r;
   n= lx_interval(sixHundredMillion,"[1,1]");
   //n now equals 10^600.000.000

   r= xp1_pow_y(1/n,n);
   cout << "(1+1/n)^n: " << r << endl;
   cout << "with Relative diameter: " << RelDiam(r) << endl;

   stagprec= 39;
   //use predefined constant E_lx_interval() for e
   lx_interval EulerNumber(E_lx_interval());

   n= lx_interval(300, "[1,1]"); //10^300
   r= pow(1+1/n, n);
   cout << "Enclosure of displacement e-r: ";
   cout << EulerNumber-r << endl;
   return 0;
}
```

The output produced by this C-XSC program is

```
(1+1/n)^n: {-1, [  27.1828182845,  27.1828182846]}
   with Relative diameter: 3.4128669415E-0080
Enclosure of displacement e-r:
   {-300, [   1.3591409142,   1.3591409143]}
```

Thus, using `stagprec` = 5 gives an enclosure of the result accurate to about 80 decimal digits. The function `xp1Pow(x, n)` computes $(x+1)^n$. It should be used if x is very close to 1. Increasing the length of the staggered numbers involved (i.e. simply increasing the value of the variable `stagprec`) increases the accuracy of the enclosure to a several hundred correct digits. Note that a corresponding calculation cannot be done using the computer algebra systems Mathematica or Maple. Both systems will fail!

We can also see from the output that, as expected, the value of $(1+\frac{1}{1e300})^{1e300}$ is smaller than the Euler number and that more than 300 leading digits agree in both numbers.

5.3 Verified Root Enclosures of Complex Functions

We consider a complex (interval) function. Finding its roots is a non trivial problem in numerical mathematics. Most algorithms deliver only an approximation to the exact zero without or with only a weak statement concerning its accuracy.

In this section, we describe an algorithm which allows the computation of verified enclosures of a root starting with an appropriate approximation. The algorithm used is based on the following Theorem 1 [9, 19], where $\underline{\cup}$ denotes the convex hull of two complex sets, $A \overset{\circ}{\subset} B$ means that A is fully contained in the interior of B, and $[S]$ is the interval hull of a bounded complex set S.

Theorem 1. *Let Z be a complex interval, $z \in \mathbb{C}$, $0 \neq c \in \mathbb{C}$, and let $f : \mathbb{C} \to \mathbb{C}$ be an analytic function in the convex region $R \supset z \underline{\cup} (z + Z)$. If*

$$-c \cdot f(z) + [(1 - c \cdot f'(z \underline{\cup} (z + Z))) \cdot Z] \overset{\circ}{\subset} Z \tag{18}$$

holds true, then there exists a unique zero z_0 of $f(z)$, with $z_0 \in z + Z$.
The proof of this theorem is given in the appendix on page 20 (see also [19]). In practice, the constant $c \neq 0$ is often set to $c = 1/f'(z)$.

In the following example we have calculated $f'(Z)$ by symbolic differentiation. This sometimes tedious procedure can be replaced by using automatic differentiation, which delivers enclosures $f(Z)$ as well as $f'(Z)$ simultaneously (see [14, 13]).
The following numerical example is taken from [19]. We consider the function

$$f(z) = \arctan\big((z-a)\cdot\ln(z^2 - 5\cdot z + 8 + i)\big) \text{ with first derivative}$$

$$f'(z) = \frac{\ln(z^2 - 5\cdot z + 8 + i) + \dfrac{(z-a)\cdot(2\cdot z - 5)}{z^2 - 5\cdot z + 8 + i}}{1 + (z-a)^2 \cdot (\ln(z^2 - 5\cdot z + 8 + i))^2.}$$

Then, setting $a = 4i$ the zeros of $f(z)$ are

$$z_0 = 2 + i, \qquad z_1 = 3 - i, \qquad z_2 = 4i.$$

With a close enough approximation to z_0 the following C-XSC program calculates guaranteed enclosures of the root z_0.

```
#include <iostream>
using namespace std;

#include "../lx_cimath.hpp" //extended staggered complex intervals
using namespace cxsc;

static const lx_cinterval c(8,1); // constant value 8+i
static const lx_cinterval a(0,4); // parameter value a = 4*i

//complex function f(z) to be considered
//(its zeros are a, 2+i, and 3-i):
lx_cinterval f(const lx_cinterval &z) {

   return atan( (z-a)*ln(sqr(z)-5*z+c) );
}

//derivative df(z) of the function f(z):
lx_cinterval df(const lx_cinterval &z) {
   lx_cinterval y, w;
   y= ln(sqr(z)-5*z+c);
   w= y + (z-a)*(2*z-5)/( sqr(z)-5*z+c );
   w= w/( 1 + sqr(z-a)*sqr(y) );
   return w;
```

```
}

int main() {

  stagprec = 2;

  //conrol output format
  cout << SetDotPrecision(11*stagprec+4,11*stagprec);
  cout << boolalpha;

  //extended complex staggered interval variable
  lx_cinterval xk;

  //let us see whether we can compute a verified enclosure
  //for the zero 2+i of f(z) with high accuracy.
  //Take as complex starting interval xk a wide enclosure of 2+i
  xk= lx_cinterval(interval(1.997,2.007),interval(0.996,1.0091));
  cout << "Iterate " << 0 << ": " << xk << endl;

  lx_cinterval m, dfxk, Nxk, xkp1;
  bool verified(false);

  for (int i=1; i<7; i++) {
     m= mid(xk);          //midpoint
     dfxk= df(xk);        //derivative evaluated over xk

     Nxk= m - f(m)/dfxk;  //Newton operator is applied

     //compute new iterate as the intersection of the result of the
     //inteval Newton operator and the former iterate:
     xkp1= Nxk & xk;      //new iterate
     cout << "Iterate " << i << ": " << xkp1 << endl;
     verified|= (Nxk < xk);
     cout << "Verification successful: " << verified << endl;

     if (xk == xkp1) break; //no further improvment
     xk= xkp1;             //replace old iterate by new iterate
  }

  return 0;
}

/* Generated output:

Iterate 0: ({-1, [  19.969999999999999886313162278,
                    20.070000000000000383693077311]},
                         {-1, [   9.959999999999999907629444351,
                                 10.091000000000000285638179776]})
Iterate 1: ({-1, [  19.969999999999999886313162278,
                    20.053435617362364240534589 05]},
```

```
                        {-1, [   9.95999999999999907629444351,
                                10.02395086899697496107819462]})
Verification successful: false
Iterate 2: ({-1, [  19.97181867684503586701794120,
                    20.00826596837999815647890501]},
                        {-1, [   9.99327987289646024748890340,
                                10.02383131624564427397672262]})
Verification successful: true
Iterate 3: ({-1, [  19.99677979348013678873030585,
                    20.00471667293638944329359220]},
                        {-1, [   9.99541643106646127137082658,
                                10.00306264985216309071347496]})
Verification successful: true
Iterate 4: ({-1, [  19.99995051088503572600529878,
                    20.00004615150148978841571080]},
                        {-1, [   9.99995043206047995454355259,
                                10.00005271746818813483059785]})
Verification successful: true
Iterate 5: ({-1, [  19.99999999880470014090860786,
                    20.00000000119496945671926369]},
          {-1, [   9.99999999878747303228010423,
                                 10.00000000121111654038941197]})
Verification successful: true
Iterate 6: ({-1, [  19.99999999999999999999411052,
                    20.00000000000000000000588949]},
                        {-1, [   9.99999999999999999998803765,
                                10.00000000000000000001196248]})
Verification successful: true

The progress concerning the accuracy from iterate 5 to
iterate 6 shows the expected quadratic convergence rate
of Newton's method. Increasing stagprec and performing
some more iteration steps gives an enclosure of the
zero accurate to several hundred digits.
*/
```

The algorithm works well with a sufficiently good approximation to the root. Thus, it is advisable to perform some Newton steps to get an approximation even closer to the zero. For simplicity, we omit this step. If the inclusion relation (18) is valid, further Interval-Newton steps may improve a possibble wide enclosing interval. This may be seen from the output produced by the given program.

6 Conclusion and Future Work

Extended staggered interval arithmetics are very powerful tools for reliable numerical computations far beyond IEEE double computations and/or interval computations using ordinary interval data types. We have realized such extended arithmetics for real and for complex (staggered interval) computations. For the new extended staggered data types

a complete set of transcendental elementary functions is available. The scaling factor introduced in the extended data types allows a very wide exponent range. Additionally, as our numerical exaples show, the accuracy of numerical computations is improved significantly compared to ordinary staggered interval computations. The improvement results mainly from our automatic scalings introduced in the implementation of the basic arithmetical extended staggered operations.

The new arithmetics would be very fast if an exact dot product for vectors with floating point components was supported by hardware. Exact dot products are a key feature to get numerical results with high accuracy. Therfore, we ask the manufactures of processors to incorporate an exact dot product facility on their chips! The (little) additional amount of hardware [24, 25] would be worth a mint for scientists/engineers interested in fast and highly accurate reliable numerical computations.

Our work concerning extended staggered numbers is still in progress [6]. Up to now the main focus are point computations or computations with very narrow intervals (with respect to the staggered level in use). We will support also wide interval computations. Additionally, we will replace the integer scaling factor (32 bit) by a scaling factor of type double (53 mantissa bit). Thus, the exponent range will be widened even more. Last but not least the input/output facilities of extended staggered quantities should be improved and automatic differentiation [8] should be provided.

A Proof of Theorem 1

First some notations/definitions:

- $z = x + i \cdot y \in \mathbb{C}$.
  In this appendix we use $|z| := |x| + |y|,$ with $|z| \geq \sqrt{x^2 + y^2}$.
- $S \subset \mathbb{C}$ denotes an arbitrary bounded subset of $\mathbb{C}$.
  We define $|S| := \max_{z \in S}\{|z|\}$.
  $[\, S\,]$ denotes the interval hull of S.
- With $S_1 \subset \mathbb{C}$ and $S_2 \subset \mathbb{C}$ the convex hull is denoted by $S_1 \underline{\cup} S_2$.
- The diameter of a real interval $X = [\underline{x}, \overline{x}]$ is defined by $\mathrm{d}(X) := \overline{x} - \underline{x}$.
- The diameter of a complex interval $Z = X + i \cdot Y$ is defined by $\mathrm{d}(Z) := \mathrm{d}(X) + \mathrm{d}(Y)$. It can easily be shown that $\mathrm{d}(z + Z) = \mathrm{d}(Z)$.
- $S \cdot Z := \{w \,|\, w = s \cdot z;\ s \in S, z \in Z\}$.

With the above notations it holds[3]

$$\mathrm{d}([\, S \cdot Z\,]) \geq |S| \cdot \mathrm{d}(Z). \tag{19}$$

Theorem 1 as formulated on page 17: *Let Z be a complex interval, $z \in \mathbb{C}$, $0 \neq c \in \mathbb{C}$, and let $f : \mathbb{C} \to \mathbb{C}$ be an analytic function in the convex region $R \supset z \underline{\cup} (z + Z)$, then with*

$$-c \cdot f(z) + [(1 - c \cdot f'(z \underline{\cup} (z + Z))), \cdot Z] \overset{\circ}{\subset} Z \tag{20}$$

[3] It should be noted that (19) is a generalization of $\mathrm{d}(A \cdot B) \geq |A| \cdot \mathrm{d}(B)$, where A and B are complex intervals, [2, (15), page 72].

there exists a unique zero z_0 of $f(z)$, with $z_0 \in z + Z$.

Proof: (see also [19])

Let $f : \mathbb{C} \to \mathbb{C}$ be an analytic function in R. For the range $\mathrm{h}(z + Z)$ of the analytic auxiliary function

$$\mathrm{h}(z) := z - c \cdot f(z) \tag{21}$$

it holds the inclusion relation, [9, Theorem 5., page 35]

$$\mathrm{h}(z + Z) \subset \mathrm{h}(z) + [\,\mathrm{h}'(z \underline{\cup} (z + Z)) \cdot Z\,]. \tag{22}$$

With the definition of $\mathrm{h}(z)$ we find

$$\mathrm{h}(z + Z) \subset z - c \cdot f(z) + [\,(1 - c \cdot f'(z \underline{\cup} (z + Z))) \cdot Z\,], \tag{23}$$

and together with (20) we get the inclusion relation

$$\mathrm{h}(z + Z) \overset{\circ}{\subset} z + Z. \tag{24}$$

Thus, Brouwer's fixed-point theorem shows that the auxiliary function $\mathrm{h}(z)$ has at least one fixed-point $z_0 \in z + Z$, i.e. $\mathrm{h}(z_0) = z_0$, and with the definition of $\mathrm{h}(z)$ it follows that $f(z)$ has at least one zero $z_0 \in z + Z$.

We now show that this zero $z_0 \in z+Z$ is unique. The inclusion relation (20) implies

$$z - c \cdot f(z) + [(1 - c \cdot f'(z \underline{\cup} (z + Z))) \cdot Z] \overset{\circ}{\subset} Z + z,$$

and for the diameter d we get the estimations

$$\begin{aligned} \mathrm{d}(z + Z) &> \mathrm{d}(z - c \cdot f(z) + [(1 - c \cdot f'(z \underline{\cup} (z + Z))) \cdot Z]) \\ &= \mathrm{d}([(1 - c \cdot f'(z \underline{\cup} (z + Z))) \cdot Z]) \\ &\geq |1 - c \cdot f'(z \underline{\cup} (z + Z))| \cdot \mathrm{d}(Z). \end{aligned}$$

The equal sign is substantiated by a simple displacement in the z-plane, generated by $z - c \cdot f(z)$, and the last line follows with (19). If in Theorem 1 condition (20) is fulfilled, Z cannot be a point interval and so $\mathrm{d}(Z)$ has to be positive. Hence $\mathrm{d}(z+Z) = \mathrm{d}(Z) > 0$, and, with the above estimation, we get

$$|1 - c \cdot f'(z \underline{\cup} (z + Z))| < 1 \quad \text{resp.} \quad \left|\mathrm{h}'(z \underline{\cup} (z + Z))\right| < 1. \tag{25}$$

This demonstrates that the function $\mathrm{h}(z)$ is a contractive mapping in $z+Z \subset z \underline{\cup} (z+Z)$. According to Banach's fixed-point theorem exactly one fixed-point z_0 exists. Hence

$$z_0 \in z + Z, \quad \text{and} \quad \mathrm{h}(z_0) = z_0.$$

With (25) we see

$$c \neq 0 \quad \text{and} \quad 0 \notin f'(z \underline{\cup} (z + Z)).$$

Hence, from the definition of $\mathrm{h}(z)$ it follows that the unique fixed-point z_0 of $\mathrm{h}(z)$ is a unique simple root of $f(z)$ ∎

References

1. Auzinger, W. and Stetter H.J.*Accurate Arithmetic Results for Decimal Data on Non-Decimal Computers.* Computing 35, 141-151, 1985.
2. Alefeld G. and Herzberger J. *Introduction to Interval Computations.* Academic Press, New York, 1983.
3. American National Standards Institute/Institute of Electrical and Electronics Engineers: “IEEE Standard for Binary Floating-Point Arithmetic”; ANSI/IEEE Std 754–1985, New York, 1985.
4. Blomquist, F.; Hofschuster, W.; Krämer, W.: Realisierung der hyperbolischen Cotangens-Funktion in einer Staggered-Correction-Intervallarithmetik in C-XSC. Preprint 2004/3, Wissenschaftliches Rechnen/Softwaretechnologie, Universitt Wuppertal, 2004.
5. Blomquist F. Verbesserungen im Bereich komplexer Standardfunktionen, interne Mitteilung, Bergische Universität Wuppertal, 2005.
6. Blomquist, F.; Hofschuster, W.; Krämer, W.: Real and Complex Staggered (Interval) Arithmetics with Wide Exponent Range (in German). Preprint 2008/1, Scientific Computing/Software Engineering, University of Wuppertal, 2008.
7. Blomquist, F.; Hofschuster, W.; Krämer, W., Neher, M.: Complex Interval Functions in C-XSC, Preprint BUW-WRSWT 2005/2, Scientific Computing/Software Engineering, University of Wuppertal, pp. 1-48, 2005.
8. Blomquist, F.; Hofschuster, W.; Krämer, W.: Real and Complex Taylor Arithmetic in C-XSC. Preprint BUW-WRSWT 2005/4, Universitt Wuppertal, 2005.
9. Böhm, H.: Berechnung von Polynomnullstellen und Auswertung arithmetischer Ausdrücke mit garantierter maximaler Genauigkeit. Dissertation, Universität Karlsruhe, 1983.
10. Bohlender, G.: What Do We Need Beyond IEEE Arithmetic? In: Ullrich, Ch. (ed): Computer Arithmetic and Self-Validating Numerical Methods, Academic Press, 1990.
11. Braune K., Krämer W. *High Accuracy Standard Functions for Real and Complex Intervals.* In Kaucher E., Kulisch U. and Ullrich Ch., editors, Computerarithmetic: Scientific Computation and Programming Languages, pages 81-114. Teubner, Stuttgart, 1987.
12. Braune K. Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy. *Computing Supplementum*, 6:159-184, 1988.
13. Griewank, A., Corliss, G. (Eds): Automatic Differentiation of Algorithms: Theory, Implementation, and Applications. SIAM, Philadelphia, Pennsylvania,1991.
14. Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D., C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer-Verlag, Berlin / Heidelberg / New York, 1995.
15. Herzberger, J. (Ed), Topics in Validated Computations. Proceedings of IMACS-GAMM International Workshop on Validated Numerics, Oldenburg, 1993. North Holland, 1994.
16. Hofschuster, W., Krämer, W.: C-XSC – A C++ Class Library for Extended Scientific Computing. In *Numerical Software with Result Verification.* R. Alt, A. Frommer, B. Kearfott, W. Luther (eds), Springer Lecture Notes in Computer Science, pp. 15 - 35, 2004.
17. Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: C-XSC, A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Berlin / Heidelberg / New York, 1993.
18. Klotz, G.: Faktorisierung von Matrizen mit maximaler Genauigkeit. Dissertation, Universität Karlsruhe, 1987.
19. Krämer, W.: Inverse Standardfunktionen für reelle und komplexe Intervallargumente mit a priori Fehlerabschätzungen für beliebige Datenformate, Dissertation, Universität Karlsruhe, 1987.
20. Krämer, W.: Mehrfachgenaue reelle und intervallmäßige Staggered-Correction Arithmetik mit zugehörigen Standardfunktionen, Bericht des Instituts für Angewandte Mathematik, Universität Karlsruhe, S. 1-80, 1988.

21. Krämer, W. and Walter, W.: FORTRAN-SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH, General Information Notes and Sample Programs. IBM DeutschlandGmbH, 1989.
22. Krämer, W.; Kulisch, U.; Lohner, R.: Numerical Toolbox for Verified Computing II. Springer Verlag, to appear.
23. Kulisch, U.: The Fifth Floating-Point Operation for Top-Performance Computers or Accumulation of Floating-Point Numbers and Products in Fixed-Point Arithmetic. Bericht 4/1997 des Forschungsschwerpunkts Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, Universität Karlsruhe (1997).
24. Kulisch, U.: Advanced Arithmetic for the Digital Computer. Design of Arithmetic Units. Springer Verlag, Wien, 2002.
25. Kulisch, U.: Computer Arithmetic and Validity – Theory, Implementation and Application. To appear 2008.
26. Lohner, R.: Einschließung der Lösung gewöhnlicher Anfangs- und Rndwertaufgaben mit Anwendungen. Dissertation, Universitt Karlsruhe, 1988.
27. Lohner, R.: Enclosing all Eigenvalues of Symmetric Matrices. In: Ullrich, Ch. and Wolff von Gudenberg, J. (eds): Accurate Numerical Algorithms, Research Reports ESPRIT, Springer, Berlin, Heidelberg, New York, 1989.
28. Lohner, R.: Interval arithmetic in staggered correction format. In: Adams, E., Kulisch, U.(Eds): Scientific Computing with Automatic Result Verification. Academic Press, San Diego, pp 301-321, 1993.
29. Neher, M.: Complex Standard Functions and Their Implementation in the CoStLy Library, ACM Transactions on Mathematical Software, Vol. 33, Number 1, 27 pages, 2007.
30. Rotmaier, B.: Die Berechnung der elementaren Funktionen mit beliebiger Genauigkeit. Dissertation, Universität Karlsruhe, 1971.
31. Rump S. M.: Kleine Fehlerschranken bei Matrixproblemen. Dissertation, Universität Karlsruhe, 1980.
32. Stetter, H.J. *Staggered Correction Representation, a Feasible Approach to Dynamic Precision*, Proceedings of the Symposium on Scientific Software, edited by Cai, Fosdick, Huang, China University of Science and Technology Press, Beijing, China, 1989.
33. Toussaint, F.: Implementierung reeller und intervallmäßiger Standardfunktionen für eine staggered-correction Langzahlarithmetik in C-XSC. Diplomarbeit, Institut für Angewandte Mathematik, Universität Karlsruhe, 1993.
34. XSC website on programming languages for scientific computing with validation. `http://www.xsc.de`